LFSC for SMT Proofs: Work in Progress

Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen,
Harley Eades, Corey Oliver, Ruoyu Zhang

The University of Iowa

Abstract

This paper presents work in progress on a new version, for public release, of the Logical Framework with Side Conditions (LFSC), previously proposed as a proof meta-format for SMT solvers and other proof-producing systems. The paper reviews the type-theoretic approach of LFSC, presents a new input syntax which hides the type-theoretic details for better accessibility, and discusses work in progress on formalizing and implementing a revised core language.

1 LFSC and the Challenge of SMT Proofs

The recent widespread adoption of SMT solvers for applications in program analysis, hardware and software verification, and others, has been enabled largely by the SMT community's successful adoption of first the SMT-LIB 1.x and now the SMT-LIB 2.x format [8, 1]. This is truly a laudable achievement of the whole SMT community, which has provided many benefits to applications. The community is also quite interested in doing the same for proofs produced by SMT solvers. This would enable advanced applications which need proofs from SMT solvers (e.g., [4]), and would benefit interactive theorem proving [3]. Having a common proof format and providing suitable incentives (e.g., through competition) could help increase the number of proof-producing solvers. But as the authors and others have argued, devising a common proof system suitable for all SMT solvers is a daunting task, as proof systems tend to mirror solving algorithms, which are quite diverse.

So the community has been considering different proposals for a common meta-format. Besson *et al.* have proposed a meta-format parametrized by solver-specific rules [2]. Their format provides notions of named clauses (via clause ids) and local proofs. This is quite useful, as one can expect to need these features across all particular sets of solver-specific rules. Nevertheless, their format does not provide means for formally defining the semantics of such rules. The ability to give a formal, declarative specification of solver-specific inferences serves as formal documentation, and also enables generic implementation of proof tools like proof checkers or proof translators (to interactive theorem provers, say). We believe Besson *et al.*'s format should be compatible with what we are proposing, though despite some preliminary conversations (particularly with Pascal Fontaine) the details of this remain to be worked out.

In previous papers, we proposed a version of the Edinburgh Logical Framework (LF) [5], extended with support for computational side conditions on inference rules, as a meta-format for SMT [6, 12]. This Logical Framework with Side Conditions (LFSC) combines powerful features of LF (reviewed below) with support for defining side conditions as recursive programs written in a simple first-order functional language. This allows proof systems to be expressed partly declaratively (with inference rules), and partly computationally (with side conditions). The advantage of using computational side conditions is that proofs written in the proof system do not contain any subproof corresponding to the side conditions. Instead, the proof checker executes the side conditions whenever it checks inferences in the proof. This can save both space and checking time. Previous work explored different balances of declarative and computational rules for the SMT-LIB logics QF_IDL and QF_LRA, and showed how to use type computation

$$\begin{array}{lcl} \textit{formulas}\ \phi & ::= & p \mid \phi_1 \to \phi_2 \\ \textit{contexts}\ \Gamma & ::= & \cdot \mid \Gamma, \phi \end{array}$$

$$\dfrac{\phi \in \Gamma}{\Gamma \vdash \phi}\ \textit{Assump} \qquad \dfrac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \to \phi_2}\ \textit{ImpIntro} \qquad \dfrac{\Gamma \vdash \phi_1 \to \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}\ \textit{ImpElim}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot,(p \to (q \to r)),q,p \vdash p \to (q \to r)} \quad \dfrac{}{\cdot,(p \to (q \to r)),q,p \vdash p}}{\cdot,(p \to (q \to r)),q,p \vdash q \to r} \quad \dfrac{}{\cdot,(p \to (q \to r)),q,p \vdash q}}{\cdot,(p \to (q \to r)),q,p \vdash r}}{\cdot,(p \to (q \to r)),q \vdash (p \to r)}}{\cdot,(p \to (q \to r)) \vdash (q \to (p \to r))}}{\cdot \vdash (p \to (q \to r)) \to (q \to (p \to r))}$$

Figure 1: Standard mathematical definition of minimal propositional logic, and example proof

for LFSC to compute interpolants [10, 9]. Optimized LFSC proof checking is quite efficient relative to solving time: for QF_UF benchmarks we found a total average overhead of 30% for proof generation and checking (together) over and above solving alone.

We are now focusing on providing a high-quality implementation of LFSC for public release. It is our hope that the availability of such an implementation will be the final step which will enable solver implementors to adopt LFSC as a single, semantics-based meta-format for SMT proofs. In the rest of this short paper, we describe our work in progress towards this vision, which consists of devising a more accessible input syntax (Section 2) and formalizing a new core language which is able to accommodate features like local definitions and implicit arguments (Section 3), which we found to be important in practice but which are often not considered in standard treatments of LF.

We would like to acknowledge also the contributions of Jed McClurg and Cuong Thai to earlier work on this new version of LFSC.

2 A Surface Syntax Based on Grammars and Rules

LF and LFSC are meta-languages based on a dependent type theory. The methodology for using LF/LFSC is to encode object-language constructs using dependently typed data constructors in LF/LFSC. If used directly, this requires unfamiliar type-theoretic notation and conceptualization, a burden we do not wish to impose on SMT solver implementors.

To illustrate these points, let us consider an example proof system in standard mathematical notation. Figure 1 shows the syntax and inference rules of minimal propositional logic. As our proposed syntax for side-condition code has not changed fundamentally, we are deliberately restricting our attention here to a signature without side conditions. For more on side-condition code in LFSC, including examples for a resolution calculus, see [6].

In the LF methodology, a proof system like the one in Figure 1 is encoded as a sequence of type declarations called a signature. The syntactic categories, here *Formula* and *Context*, are encoded as types, whose syntactic constructs are encoded as constructors of those types. The judgments of the proof system are also encoded as types, with the inference rules as

$$\begin{array}{lcl}
\mathit{formula} & : & \mathbf{Type} \\
\mathit{imp} & : & \mathit{formula} \to \mathit{formula} \to \mathit{formula} \\
\mathit{holds} & : & \mathit{formula} \to \mathbf{Type} \\
\mathit{ImpIntro} & : & \Pi f_1 : \mathit{formula}.\Pi f_2 : \mathit{formula}.((\mathit{holds}\ f_1) \to (\mathit{holds}\ f_2)) \to (\mathit{holds}\ (\mathit{imp}\ f_1\ f_2)) \\
\mathit{ImpElim} & : & \Pi f_1 : \mathit{formula}.\Pi f_2 : \mathit{formula}.(\mathit{holds}\ (\mathit{imp}\ f_1\ f_2)) \to (\mathit{holds}\ f_1) \to (\mathit{holds}\ f_2)
\end{array}$$

$$\begin{array}{l}
\mathit{ImpIntro}\ (\mathit{imp}\ p\ (\mathit{imp}\ q\ r))\ (\mathit{imp}\ q\ (\mathit{imp}\ p\ r)) \\
\quad \lambda u.\mathit{ImpIntro}\ q\ (\mathit{imp}\ p\ r) \\
\quad\quad \lambda v.\mathit{ImpIntro}\ p\ r \\
\quad\quad\quad \lambda w.\mathit{ImpElim}\ q\ r\ (\mathit{ImpElim}\ p\ (\mathit{imp}\ q\ r)\ u\ w)\ v
\end{array}$$

Figure 2: Minimal propositional logic in LF (core syntax), with encoded example proof (no inferred arguments)

constructors. Object-language variables are represented with meta-language variables, object-language binders with LF's λ-binder, and logical contexts (like Γ in this example) with the LF typing context.

Figure 2 shows the corresponding signature in standard type-theoretic notation for LF. The *Assump* rule does not have a constructor, because assumptions in the object language have been mapped to variables in LF. For the example proof, the λ-bound variables u, v, and w correspond to the assumptions of $p \to (q \to r)$, q, and p, respectively. Since inference rules are encoded using term constructors with dependent types (Π types), there is rather a lot of redundant information in that encoded example proof. Implementations of LF like Twelf allow one to omit some of these arguments in some cases, if they can be inferred from other parts of the proof term [7]. In our first implementation of LFSC, we allowed proofs to contain underscores for omitted arguments, which we then sought to reconstruct in a similar way to what is done in Twelf.

We are designing a surface language for LFSC, intended to rely on the more familiar concepts of context-free grammars for describing syntax, and more familiar notation for inference rules, as is done in tools like Ott (and others) [11]. Using this surface language, our sample signature is encoded as in Figure 3. The `ImpIntro` rule uses square brackets and a turnstile for hypothetical reasoning; in this case, to represent the fact that `holds f2` is to be proved under the assumption of `holds f1`. We use root names like `f` for `formula` to avoid specifying types for meta-variables in the rules. Root names are specified when a syntactic category is defined. The example proof term is similar to the one in Figure 2, except that arguments which an analysis determines can be reconstructed can be omitted. Defining this analysis is still work in progress, but in this case, it will determine that of the meta-variables used in the `ImpIntro` and `ImpElim` rules, only `f1` needs a value in order to guarantee that the types computed for terms are fully instantiated (that is, without free meta-variables). The benefits of omitting arguments in dependently typed languages are well known: notice how much more compact the example proof term is without values for the meta-variables `f2` for `ImpIntro` and `f1` and `f2` for `ImpElim`.

For encoding SMT logics, we made use, in our previous proposals, of LF type-checking to implement SMT type-checking of SMT terms contained in proofs. This is done in a standard way by using an indexed type `term T` to represent SMT terms which have (encoded) SMT type `T`. Our new surface syntax also supports indexed syntactic categories. For example, Figure 4 gives part of a signature for a minimal SMT theory. This syntax definition indexes the syntactic

```
  SYNTAX
  formula f ::= imp f1 f2 .

  JUDGMENTS
  (holds f)

  RULES

  [ holds f1 ] |- holds f2
  ----------------------------  ImpIntro
  holds (imp f1 f2) .

  holds (imp f1 f2) ,  holds f1
  ----------------------------  ImpElim
  holds f2 .


ImpIntro (imp p (imp q r))
   u. ImpIntro q
      v. ImpIntro p
        w. ImpElim (ImpElim u w) v
```

Figure 3: LFSC encoding of minimal propositional logic in proposed surface syntax, with example proof

```
  SYNTAX
  sort s ::= arrow s1 s2 | bool .
  term<sort> t ::=
                true<bool>
              | (not t1<bool>)<bool>
              | (impl t1<bool> t2<bool>)<bool>
              | (forall t<s> ^ t<bool>)<bool>
              | (ite t1<bool> t2<s> t3<s>)<s>
              | (eq t1<s> t2<s>)<bool>.
  formula f ::= t<bool>.
```

Figure 4: LFSC encoding (surface syntax) of part of the syntax of SMT terms, indexed by their sorts

category `term` by the syntactic category `sort`. Everywhere a meta-variable for terms is used, it must be listed with its sort. Similarly, the definitions of constructors show the resulting indices. For example, since equality is sort-polymorphic, we see that its meta-variables are indexed by (the same) sort `s`, while its resulting sort is indicated as `bool`. This example also demonstrates our surface syntax for binding, namely `^`, in the declaration of `forall`. This notation means that `forall` binds a variable ranging over terms of sort `s`, and then has a body which must be a term of sort `bool`.

As a final example of the new syntax, Figure 5 shows the LFSC encodings of SMT rules for universal quantifiers. The rules are complicated somewhat by the need to express the sort information for terms. The first rule uses the `^` binding notation to indicate a parametric judgment in the premise: to apply `all_intro`, we must supply a proof of `holds f{t<s>}`

```
t<s> ^ holds f { t<s> }
-------------------------------- all_intro
holds (forall t<s> ^ f{t<s>} ).

holds (forall t<s> ^ f { t<s> } )
---------------------------------- all_elim
holds f{t<s>} .
```

Figure 5: LFSC encoding (surface syntax) of SMT quantifier rules

for an arbitrary term `t` of sort `s`. The fact that the term is arbitrary is indicated by the use of `^` in the premise. We also see a notation for expression contexts, using curly braces. Any meta-variable can be used as a context variable, although if that meta-variable is from an indexed syntactic category, the result index must also be shown. Here, we make use of a defined syntactic category of formulas, so that no result sort need be listed for meta-variable `f`. All these different binding concepts (expression contexts, parametric proofs, and hypothetical proofs) are ultimately implemented with just the single λ-binder of LF. But for the surface language, our common informal meta-language tends to distinguish them, and so we follow that approach in our surface language.

3 Improving the Core Language

While the surface language distinguishes syntactic constructs and proof rules, LF elegantly unites them (under encoding) as just term constructors for possibly indexed datatypes. So both for conciseness and to give a semantics for the different features sketched above, we are working to compile signatures from our surface language to a core language for LFSC, which we are specifying formally with the help of the Ott tool [11]. We are addressing a number of issues important for practical use of LF/LFSC for large proofs:

- We have explicit syntax to indicate which arguments to term constructors are implicit and which are explicit.
- We are devising an algorithm to ensure that all higher-order implicit arguments (which our surface language limits to second order) can be automatically reconstructed during proof checking. Other approaches simply delay constraints falling outside the decidable higher-order pattern fragment in the hopes of not needing to solve them.
- In general, we only intend to execute side-condition code on concrete terms. So we are designing the language to help ensure that as much as statically possible, the arguments to side condition programs do not contain meta-variables for omitted implicit arguments. Such meta-variables can be filled in by type checking, but not during execution of side-condition code.
- We have support for global and local definitions, both of which are incorporated into definitional equality. Some other treatments of LF do not handle definitions at all, but these are essential for keeping proof sizes down.
- Since we use definitions, we cannot so easily make use of the so-called "canonical forms" version of LF, where all terms are assumed to be in β-short η-long normal form: appli-

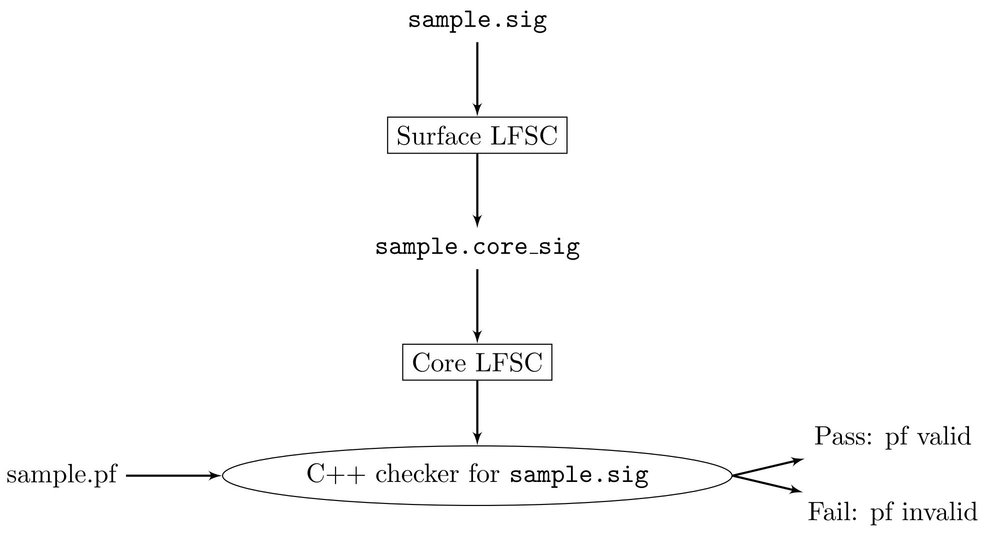

Figure 6: System architecture for new implementation of LFSC

cations of defined symbols can be redexes. Instead, we are formulating a system with explicit definitional equality, which can also fill in implicit arguments.

- We retain support for compiling proof rules to efficient low-level (C/C++) code by avoiding explicit substitutions in our formalization. Meta-level substitutions will be carried out directly in code emitted by the compiler, so using explicit substitutions merely obscures what will happen in compiled code.
- We are designing the language for side-condition programs to respect dependent typing of terms. This does not require implementing a full-blown dependently typed programming language, because indices to types can only contain terms which normalize to constructor terms. (In contrast, in dependently typed programming, indices can include terms with recursion.)

While none of these features is terribly complex, designing detailed typing rules that accommodate them all simultaneously has taken time. For example, we are implementing a requirement that all omitted arguments to a term constructor C must be reconstructed by the time we complete the type checking of an application of C. For this, we constrain the syntax of types for term constructors so that implicit arguments are listed first, then explicit arguments and possible invocations of side-condition code, and finally a possibly indexed result type.

4 Status and Conclusion

Figure 6 shows the architecture for the LFSC system we are currently implementing. A sample signature `sample.sig` in the surface syntax discussed above is translated to a core-language signature `sample.core_sig`, which is then compiled to C++ code optimized for checking proofs in that signature. A sample proof `sample.pf` can then be fed to that checker, which reports whether the proof passes (all inferences correct and all side conditions succeed) or fails.

Co-author Corey Oliver has completed the translator for surface-language signature ("Surface LFSC" in Figure 6). Co-author Austin Laugesen has completed implementation of the compiler for side-condition code (part of LFSC core in Figure 6). This compiler translates the LFSC side-condition language to C++, and supports both reference counting or regions for managing memory allocated during execution of side-condition code. A preliminary evaluation confirms the performance benefits for this application of region-based memory management, where all new memory is allocated from a single monolithic region of memory which can then be reclaimed at once in constant time. Co-authors Harley Eades and Ruoyu Zhang have worked with Aaron Stump on the design of the core language, which is almost complete. The first three co-authors have worked to devise the surface syntax, which is complete now. Implementation of the core-language type checker and compiler ("Core LFSC" in Figure 6) in OCaml are just beginning, and should be done Summer 2012. We expect to have an initial public release of the tool in early Fall 2012. We hope that this release will be a decisive step in the quest for a standard meta-format for SMT proofs, and will help the community continue its remarkable success in providing high-performance, usable logic solvers for advanced applications in many fields of Computer Science.

References

[1] C. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard: Version 2.0*, 2010. available from `www.smtlib.org`.

[2] F. Besson, P. Fontaine, and L. Théry. A Flexible Proof Format for SMT: a Proposal. In P. Fontaine and A. Stump, editors, *Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.

[3] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[4] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Programming Language Design and Implementation (PLDI)*, pages 412–423. ACM, 2010.

[5] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In B. Dutertre and O. Strichman, editors, *International Workshop on Satisfiability Modulo Theories*, 2009.

[7] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[8] S. Ranise and C. Tinelli. The SMT-LIB Standard, Version 1.2, 2006. Available from the "Documents" section of http://www.smtlib.org.

[9] Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with LFSC. In A. Gupta and D. Kroening, editors, *International Workshop on Satisfiability Modulo Theories*, 2010.

[10] Andrew Reynolds, Cesare Tinelli, and Liana Hadarean. Certified interpolant generation for EUF. In S. Lahiri and S. Seshia, editors, *International Workshop on Satisfiability Modulo Theories*, 2011.

[11] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[12] A. Stump. Proof checking technology for satisfiability modulo theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.